Sources of Optimization

• In order to optimize our IR, we need to understand why it can be improved in the first place.

• **Reason one:** IR generation introduces redundancy.
  • A naïve translation of high-level language features into IR often introduces subcomputations.
  • Those subcomputations can often be sped up, shared, or eliminated.

• **Reason two:** Programmers are lazy.
  • Code executed inside of a loop can often be factored out of the loop.
  • Language features with side effects often used for purposes other than those side effects.

Alex Aiken, Stanford
Optimizations from IR Generation

int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
Optimizations from IR Generation

```plaintext
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

```plaintext
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;

_t2 = x + x;
_t3 = y;
b2 = _t2 == _t3;

_t4 = x + x;
_t5 = y;
b3 = _t5 < _t4;
```

Alex Aiken, Stanford
Optimizations from IR Generation

```c
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

```c
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;

_t2 = x + x;
_t3 = y;
b2 = _t2 == _t3;

_t4 = x + x;
_t5 = y;
b3 = _t5 < _t4;
```

Alex Aiken, Stanford
Optimizations from IR Generation

```
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

```
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;

b2 = _t0 == _t1;

b3 = _t0 < _t1;
```

Alex Aiken, Stanford
while (x < y + z) {
    x = x - y;
}

Alex Aiken, Stanford
Optimizations from Lazy Coders

```c
while (x < y + z) {
    x = x - y;
}
```

```
_L0:
    _t0 = y + z;
    _t1 = x < _t0;
    IfZ _t1 Goto _L1;
    x = x - y;
    Goto _L0;

_L1:
```
while (x < y + z) {
    x = x - y;
}

_L0:
_t0 = y + z;
_t1 = x < _t0;
IfZ _t1 Goto _L1;
x = x - y;
Goto _L0;

_L1:
Optimizations from Lazy Coders

```c
while (x < y + z) {
    x = x - y;
}
```

Alex Aiken, Stanford
Optimizations from Lazy Coders

while (x < y + z) {
  x = x - y;
}

_t0 = y + z;
_L0:
  _t1 = x < _t0;
  IfZ _t1 Goto _L1;
  x = x - y;
  Goto _L0;
_L1:
A Note on Terminology

- The term “optimization” implies looking for an “optimal” piece of code for a program.
- This is, in general, undecidable.
  - e.g. create a program that can be simplified iff some other program halts.
- Our goal will be IR *improvement* rather than IR *optimization*.

Alex Aiken, Stanford
The Challenge of Optimization

• A good optimizer
  • Should never change the observable behavior of a program.
  • Should produce IR that is as efficient as possible.
  • Should not take too long to process inputs.

• Unfortunately:
  • Even good optimizers sometimes introduce bugs into code.
  • Optimizers often miss “easy” optimizations due to limitations of their algorithms.
  • Almost all interesting optimizations are \( \text{NP} \)-hard or undecidable.
What are we Optimizing?

- Optimizers can try to improve code usage with respect to many observable properties.
- What are some quantities we might want to optimize?

Alex Aiken, Stanford
What are we Optimizing?

- Optimizers can try to improve code usage with respect to many observable properties.
- What are some quantities we might want to optimize?
- **Runtime** (make the program as fast as possible at the expense of time and power)
- **Memory usage** (generate the smallest possible executable at the expense of time and power)
- **Power consumption** (choose simple instructions at the expense of speed and memory usage)
- Plus a lot more (minimize function calls, reduce use of floating-point hardware, etc.)

Alex Aiken, Stanford
Basic Blocks

- A **basic block** is a sequence of IR instructions where
  - There is exactly one spot where control enters the sequence, which must be at the start of the sequence.
  - There is exactly one spot where control leaves the sequence, which must be at the end of the sequence.
- Informally, a sequence of instructions that always execute as a group.

Alex Aiken, Stanford
**Control-Flow Graphs**

- A **control-flow graph** (CFG) is a graph of the basic blocks in a function.
  - The term CFG is overloaded – from here on out, we'll mean “control-flow graph” and not “context-free grammar.”
  - Each edge from one basic block to another indicates that control can flow from the end of the first block to the start of the second block.
  - There is a dedicated node for the start and end of a function.
Types of Optimizations

- An optimization is **local** if it works on just a single basic block.
- An optimization is **global** if it works on an entire control-flow graph.
- An optimization is **interprocedural** if it works across the control-flow graphs of multiple functions.
  - We won't talk about this in this course.

Alex Aiken, Stanford
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}

Alex Aiken, Stanford
```c
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```
int main() {
  int x;
  int y;
  int z;

  y = 137;
  if (x == 0)
    z = y;
  else
    x = y;
}

```plaintext
Local Optimizations

\[
\text{start}
\]

\[
\text{y = 137; IfZ x Goto _L0;}
\]

\[
\text{\_t1 = y; _t1 = z;}
\]

\[
\text{\_t2 = y; _t2 = x;}
\]

end
```
Local Optimizations

```c
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```

Alex Aiken, Stanford
Local Optimizations

```c
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```

Alex Aiken, Stanford
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}

Local Optimizations

Alex Aiken, Stanford
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
    
}
Local Optimizations

```c
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```

Alex Aiken, Stanford
Global Optimizations

int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}

Alex Aiken, Stanford
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}

Alex Aiken, Stanford
Global Optimizations

int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}

Alex Aiken, Stanford
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}

Alex Aiken, Stanford
Local Optimizations
Common Subexpression Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
Common Subexpression Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

.tmp0 = 4;
PushParam _tmp0;
.tmp1 = LCall _Alloc;
PopParams 4;
.tmp2 = Object;
*(_tmp1) = _tmp2;
x = _tmp1;
.tmp3 = 4;
a = _tmp3;
.tmp4 = a + b;
c = _tmp4;
.tmp5 = a + b;
.tmp6 = *(x);
.tmp7 = *(._tmp6);
PushParam _tmp5;
PushParam _tmp5;
ACall _tmp7;
PopParams 8;
Common Subexpression Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*( _tmp1 ) = _tmp2;
x = _tmp1;
_tmp3 = 4;
a = _tmp3;
_tmp4 = a + b;
c = _tmp4;
_tmp5 = a + b;
_tmp6 = *(x);
_tmp7 = *( _tmp6 );
PushParam _tmp5;
PushParam x;
ACall _tmp7;
PopParams 8;
Common Subexpression Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
_(*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = *(x) ;
_tmp7 = *( _tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
Common Subexpression Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;
x = _tmp1;
_tmp3 = 4;
a = _tmp3;
_tmp4 = a + b;
c = _tmp4;
_tmp5 = _tmp4;
_tmp6 = *(x);
_tmp7 = *(tmp6);
PushParam _tmp5;
PushParam x;
ACall _tmp7;
PopParams 8;
Common Subexpression Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

__tmp0 = 4;
PushParam __tmp0;
__tmp1 = LCall __Alloc;
PopParams 4;
__tmp2 = Object;
*(__tmp1) = __tmp2;
x = __tmp1;
__tmp3 = __tmp0;
a = __tmp3;
__tmp4 = a + b;
c = __tmp4;
__tmp5 = __tmp4;
__tmp6 = *(x);
__tmp7 = *(__tmp6);
PushParam __tmp5;
PushParam x;
ACall __tmp7;
PopParams 8;

Alex Aiken, Stanford
Common Subexpression Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = a + b;
c = _tmp4;
_tmp5 = _tmp4;
_tmp6 = *(x);
_tmp7 = *(tmp6);
PushParam _tmp5;
PushParam x;
ACall _tmp7;
PopParams 8;

Alex Aiken, Stanford
Common Subexpression Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = a + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = *(x);
_tmp7 = *(tmp6);
PushParam _tmp5;
PushParam x;
ACall _tmp7;
PopParams 8;

Alex Aiken, Stanford
Common Subexpression Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

// Code for the expression:
_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(__tmp1) = __tmp2;
x = __tmp1;
_tmp3 = __tmp0;
a = __tmp3;
_tmp4 = a + b;
c = __tmp4;
_tmp5 = c;
_tmp6 = *(x);
_tmp7 = *(__tmp6);
PushParam __tmp5;
PushParam x;
ACall __tmp7;
PopParams 8;
Common Subexpression Elimination

- If we have two variable assignments
  \[ v_1 = a \text{ op } b \]
  ...
  \[ v_2 = a \text{ op } b \]
and the values of \( v_1 \), \( a \), and \( b \) have not changed between the assignments, rewrite the code as
  \[ v_1 = a \text{ op } b \]
  ...
  \[ v_2 = v_1 \]
- Eliminates useless recalculation.
- Paves the way for later optimizations.

Alex Aiken, Stanford
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*( _tmp1 ) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = a + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = *( x );
_tmp7 = *( _tmp6 );
PushParam _tmp5;
PushParam x;
ACall _tmp7;
PopParams 8;
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = a + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = *(tmp1);
_tmp7 = *(tmp6);
PushParam _tmp5;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(__tmp1) = _tmp2;
x = __tmp1;
_tmp3 = __tmp0;
a = __tmp3;
_tmp4 = a + b;
c = __tmp4;
_tmp5 = c;
_tmp6 = *(__tmp1);
_tmp7 = *(__tmp6);
PushParam __tmp5;
PushParam __tmp1;
ACall __tmp7;
PopParams 8;
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

 tempList0 = 4;
PushParam tempList0;
 tempList1 = LCall _Alloc;
PopParams 4;
 tempList2 = Object;
*(tempList1) = tempList2;
x = tempList1;
 tempList3 = tempList0;
a = tempList3;
 tempList4 = tempList3 + b;
c = tempList4;
 tempList5 = c;
 tempList6 = *(tempList1);
 tempList7 = *(tempList6);
PushParam tempList5;
PushParam tempList1;
ACall tempList7;
PopParams 8;
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*( _tmp1 ) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = _tmp3 + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = *( _tmp1 );
_tmp7 = *( _tmp6 );
PushParam _tmp5;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*( _tmp1 ) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = _tmp3 + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = *( _tmp1 );
_tmp7 = *( _tmp6 );
PushParam c;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*( _tmp1 ) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = _tmp3 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = *( _tmp1 ) ;
_tmp7 = *( _tmp6 ) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(__tmp1) = _tmp2;
x = __tmp1;
_tmp3 = __tmp0;
a = __tmp3;
_tmp4 = __tmp3 + b;
c = __tmp4;
_tmp5 = c;
_tmp6 = __tmp2;
_tmp7 = *(__tmp6);
PushParam c;
PushParam __tmp1;
ACall __tmp7;
PopParams 8;
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp3;
_tmp4 = _tmp3 + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = _tmp2;
_tmp7 = *(tmp6);
PushParam c;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

___tmp0 = 4 ;
PushParam _tmp0 ;
___tmp1 = LCall _Alloc ;
PopParams 4 ;
___tmp2 = Object ;
*(___tmp1) = __tmp2 ;
x = __tmp1 ;
___tmp3 = __tmp0 ;
a = __tmp3 ;
___tmp4 = ___tmp3 + b ;
c = ___tmp4 ;
___tmp5 = c ;
___tmp6 = ___tmp2 ;
___tmp7 = *(___tmp2) ;
PushParam c ;
PushParam ___tmp1 ;
ACall ___tmp7 ;
PopParams 8 ;

Alex Aiken, Stanford
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(__tmp1) = __tmp2;
x = __tmp1;
_tmp3 = __tmp0;
a = __tmp3;
_tmp4 = __tmp3 + b;
c = __tmp4;
_tmp5 = c;
_tmp6 = __tmp2;
_tmp7 = *(__tmp2);
PushParam c;
PushParam __tmp1;
ACall __tmp7;
PopParams 8;

Alex Aiken, Stanford
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(._tmp1) = _tmp2;
x = _tmp1;
_tmp3 = _tmp0;
a = _tmp0;
_tmp4 = _tmp0 + b;
c = _tmp4;
_tmp5 = c;
_tmp6 = _tmp2;
_tmp7 = *(._tmp2);
PushParam c;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;

Alex Aiken, Stanford
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_{tmp0} = 4;
PushParam _tmp0;
_{tmp1} = LCall _Alloc;
PopParams 4;
_{tmp2} = Object;
*(_{tmp1}) = _tmp2;
x = _tmp1;
_{tmp3} = _tmp0;
a = _tmp0;
_{tmp4} = _tmp0 + b;
c = _tmp4;
_{tmp5} = c;
_{tmp6} = _tmp2;
_{tmp7} = *(_{tmp2});
PushParam c;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
_
tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*( _tmp1 ) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = c ;
_tmp6 = _tmp2 ;
_tmp7 = *( _tmp2 ) ;
PushParam c ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

��_0 = 4;
PushParam _tmp0;
_ tmp1 = LCall _Alloc;
PopParams 4;
_ tmp2 = Object;
*( _tmp1) = _ tmp2;
x = _tmp1;
_ tmp3 = 4;
a = 4;
_ tmp4 = _tmp0 + b;
c = _tmp4;
_ tmp5 = c;
_ tmp6 = _tmp2;
_ tmp7 = *( _tmp2);
PushParam c;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;

Alex Aiken, Stanford
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;
x = _tmp1;
_tmp3 = 4;
a = 4;
_tmp4 = _tmp0 + b;
c = _tmp4;
_tmp5 = _tmp4;
_tmp6 = _tmp2;
_tmp7 = *(tmp2);
PushParam _tmp4;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
Copy Propagation

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*( _tmp1 ) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = 4 ;
_tmp4 = _tmp0 + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = _tmp2 ;
_tmp7 = *( _tmp2 ) ;
PushParam _tmp4 ;
PushParam _tmp1 ;
ACall _tmp7 ;
PopParams 8 ;

Alex Aiken, Stanford
Copy Propagation

• If we have a variable assignment
  \[ v_1 = v_2 \]
  then as long as \( v_1 \) and \( v_2 \) are not reassigned, we can rewrite expressions of the form
  \[ a = \ldots v_1 \ldots \]
as
  \[ a = \ldots v_2 \ldots \]
  provided that such a rewrite is legal.

• This will help immensely later on, as you'll see.
Dead Code Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(_tmp1) = _tmp2;
x = _tmp1;
_tmp3 = 4;
a = 4;
_tmp4 = _tmp0 + b;
c = _tmp4;
_tmp5 = _tmp4;
_tmp6 = _tmp2;
_tmp7 = *(_tmp2);
PushParam _tmp4;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
Dead Code Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
Dead Code Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;

_tmp3 = 4;
a = 4;
_tmp4 = _tmp0 + b;
c = _tmp4;
_tmp5 = _tmp4;
_tmp6 = _tmp2;
_tmp7 = *(tmp2);
PushParam _tmp4;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
Dead Code Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;

_tmp3 = 4;
a = 4;
_tmp4 = _tmp0 + b;
c = _tmp4;
_tmp5 = _tmp4;
_tmp6 = _tmp2;
_tmp7 = *(tmp2);
PushParam _tmp4;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
Dead Code Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;
a = 4;
_tmp4 = _tmp0 + b;
c = _tmp4;
_tmp5 = _tmp4;
_tmp6 = _tmp2;
_tmp7 = *(tmp2);
PushParam _tmp4;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;

Alex Aiken, Stanford
Dead Code Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;

a = 4;
_tmp4 = _tmp0 + b;
c = _tmp4;
_tmp5 = _tmp4;
_tmp6 = _tmp2;
_tmp7 = *(tmp2);
PushParam _tmp4;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;

Alex Aiken, Stanford
Dead Code Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
Dead Code Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;

_tmp4 = _tmp0 + b;
c = _tmp4;
_tmp5 = _tmp4;
_tmp6 = _tmp2;
_tmp7 = *(tmp2);
PushParam _tmp4;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
Dead Code Elimination

```plaintext
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```plaintext
_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;

_tmp4 = _tmp0 + b;
_tmp5 = _tmp4;
_tmp6 = _tmp2;
_tmp7 = *(tmp2);
PushParam _tmp4;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
```
Dead Code Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```c
_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(_tmp1) = _tmp2;

_tmp4 = _tmp0 + b;
_tmp5 = _tmp4;
_tmp6 = _tmp2;
_tmp7 = *(_tmp2);
PushParam _tmp4;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
```
Dead Code Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;

_tmp4 = _tmp0 + b;

_tmp6 = _tmp2;
_tmp7 = *(tmp2);
PushParam _tmp4;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
Dead Code Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;

_tmp4 = _tmp0 + b;

_tmp6 = _tmp2;
_tmp7 = *(tmp2);
PushParam _tmp4;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;
Dead Code Elimination

Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);

_tmp0 = 4;
PushParam _tmp0;
_tmp1 = LCall _Alloc;
PopParams 4;
_tmp2 = Object;
*(tmp1) = _tmp2;

_tmp4 = _tmp0 + b;

_tmp7 = *(tmp2);
PushParam _tmp4;
PushParam _tmp1;
ACall _tmp7;
PopParams 8;

Alex Aiken, Stanford
Dead Code Elimination

- An assignment to a variable $v$ is called **dead** if the value of that assignment is never read anywhere.

- **Dead code elimination** removes dead assignments from IR.

- Determining whether an assignment is dead depends on what variable is being assigned to and when it's being assigned.
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = a + b ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
Applying Local Optimizations

- The different optimizations we've seen so far all take care of just a small piece of the optimization.
  - Common subexpression elimination eliminates unnecessary statements.
  - Copy propagation helps identify dead code.
  - Dead code elimination removes statements that are no longer needed.
- To get maximum effect, we may have to apply these optimizations numerous times.
Applying Local Optimizations

\[
b = a \times a; \\
c = a \times a; \\
d = b + c; \\
e = b + b; 
\]
Applying Local Optimizations

\[ b = a \times a; \]
\[ c = a \times a; \]
\[ d = b + c; \]
\[ e = b + b; \]
Applying Local Optimizations

\[ b = a \times a; \]
\[ c = a \times a; \]
\[ d = b + c; \]
\[ e = b + b; \]

Common Subexpression Elimination
Applying Local Optimizations

\[ b = a \times a; \]
\[ c = b; \]
\[ d = b + c; \]
\[ e = b + b; \]

Common Subexpression Elimination
Applying Local Optimizations

\[ b = a \times a; \]
\[ c = b; \]
\[ d = b + c; \]
\[ e = b + b; \]
Applying Local Optimizations

\[
\begin{align*}
\text{\scriptsize \texttt{b} } &= \text{\scriptsize \texttt{a * a;}} \\
\text{\scriptsize \texttt{c} } &= \text{\scriptsize \texttt{b;}} \\
\text{\scriptsize \texttt{d} } &= \text{\scriptsize \texttt{b + c;}} \\
\text{\scriptsize \texttt{e} } &= \text{\scriptsize \texttt{b + b;}}
\end{align*}
\]
Applying Local Optimizations

\[
b = a \times a; \\
c = b; \\
d = b + c; \\
e = b + b;
\]

Copy Propagation
Applying Local Optimizations

\[ b = a \times a; \]
\[ c = b; \]
\[ d = b + b; \]
\[ e = b + b; \]

Copy Propagation
Applying Local Optimizations

\[
b = a \times a; \\
c = b; \\
d = b + b; \\
e = b + b;
\]
Applying Local Optimizations

\[
\begin{align*}
b &= a \times a; \\
c &= b; \\
d &= b + b; \\
e &= b + b;
\end{align*}
\]
Applying Local Optimizations

\[ b = a \times a; \]
\[ c = b; \]
\[ d = b + b; \]
\[ e = b + b; \]

Common Subexpression Elimination (Again)
Applying Local Optimizations

\[
b = a \times a;
c = b;
d = b + b;
e = d;
\]

Common Subexpression Elimination (Again)

Alex Aiken, Stanford
Applying Local Optimizations

\[
b = a \ast a; \\
c = b; \\
d = b + b; \\
e = d;
\]
Other Types of Local Optimization

- **Arithmetic Simplification**
  - Replace “hard” operations with easier ones.
  - e.g. rewrite \( x = 4 \times a \) as \( x = a \ll 2 \).

- **Constant Folding**
  - Evaluate expressions at compile-time if they have a constant value.
  - e.g. rewrite \( x = 4 \times 5 \) as \( x = 20 \).
Implementing Local Optimization

Alex Aiken, Stanford
Optimizations and Analyses

- Most optimizations are only possible given some analysis of the program's behavior.
- In order to implement an optimization, we will talk about the corresponding program analyses.
Available Expressions

- Both common subexpression elimination and copy propagation depend on an analysis of the available expressions in a program.
- An expression is called **available** if some variable in the program holds the value of that expression.
- In common subexpression elimination, we replace an available expression by the variable holding its value.
- In copy propagation, we replace the use of a variable by the available expression it holds.
Finding Available Expressions

• Initially, no expressions are available.

• Whenever we execute a statement \( a = b + c \):
  • Any expression holding \( a \) is invalidated.
  • The expression \( a = b + c \) becomes available.

• **Idea**: Iterate across the basic block, beginning with the empty set of expressions and updating available expressions at each variable.

Alex Aiken, Stanford
Available Expressions

a = b;
c = b;
d = a + b;
e = a + b;
d = b;
f = a + b;
Available Expressions

{   }
a = b;

c = b;

d = a + b;

e = a + b;

f = a + b;
Available Expressions

{ }
a = b;
{ a = b }
c = b;

d = a + b;
e = a + b;

d = b;
f = a + b;
Available Expressions

{ }
 a = b;
{ a = b }
 c = b;
{ a = b, c = b }
 d = a + b;
 e = a + b;

 d = b;
 f = a + b;
Available Expressions

\{ \} 
\texttt{a = b;}
\{ \texttt{a = b} \}
\texttt{c = b;}
\{ \texttt{a = b, c = b} \}
\texttt{d = a + b;}
\{ \texttt{a = b, c = b, d = a + b} \}
\texttt{e = a + b;}

\texttt{d = b;}
\texttt{f = a + b;}

Alex Aiken, Stanford
Available Expressions

{ }
a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;

f = a + b;
Available Expressions

{ }  
\( a = b; \)
\{ a = b \}  
\( c = b; \)
\{ a = b, c = b \}  
\( d = a + b; \)
\{ a = b, c = b, d = a + b \}  
\( e = a + b; \)
\{ a = b, c = b, d = a + b, e = a + b \}  
\( d = b; \)
\{ a = b, c = b, d = b, e = a + b \}  
\( f = a + b; \)
Available Expressions

{   }
  a = b;
{   a = b   }
  c = b;
{   a = b, c = b   }
  d = a + b;
{   a = b, c = b, d = a + b   }
  e = a + b;
{   a = b, c = b, d = a + b, e = a + b   }
  d = b;
{   a = b, c = b, d = b, e = a + b   }
  f = a + b;
{   a = b, c = b, d = b, e = a + b, f = a + b   }

Alex Aiken, Stanford
Common Subexpression Elimination

{ }

a = b;
{ a = b }
c = b;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = a + b;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }

Alex Aiken, Stanford
Common Subexpression Elimination

{ }
\textbf{a} = \textbf{b};
{ \textbf{a} = \textbf{b} }
\textbf{c} = \textbf{b};
{ \textbf{a} = \textbf{b}, \textbf{c} = \textbf{b} }
\textbf{d} = \textbf{a} + \textbf{b};
{ \textbf{a} = \textbf{b}, \textbf{c} = \textbf{b}, \textbf{d} = \textbf{a} + \textbf{b} }
\textbf{e} = \textbf{a} + \textbf{b};
{ \textbf{a} = \textbf{b}, \textbf{c} = \textbf{b}, \textbf{d} = \textbf{a} + \textbf{b}, \textbf{e} = \textbf{a} + \textbf{b} }
\textbf{d} = \textbf{b};
{ \textbf{a} = \textbf{b}, \textbf{c} = \textbf{b}, \textbf{d} = \textbf{b}, \textbf{e} = \textbf{a} + \textbf{b} }
\textbf{f} = \textbf{a} + \textbf{b};
{ \textbf{a} = \textbf{b}, \textbf{c} = \textbf{b}, \textbf{d} = \textbf{b}, \textbf{e} = \textbf{a} + \textbf{b}, \textbf{f} = \textbf{a} + \textbf{b} }

Alex Aiken, Stanford
Common Subexpression Elimination

```
{   }
  a = b;
  {   a = b   }
  c = a;
  {   a = b, c = b   }
  d = a + b;
  {   a = b, c = b, d = a + b   }
  e = a + b;
  {   a = b, c = b, d = a + b, e = a + b   }
  d = b;
  {   a = b, c = b, d = b, e = a + b   }
  f = a + b;
  {   a = b, c = b, d = b, e = a + b, f = a + b   }
```

Alex Aiken, Stanford
Common Subexpression Elimination

\[
\begin{align*}
\{ \} \\
a &= b; \\
\{ a = b \} \\
c &= a; \\
\{ a = b, c = b \} \\
d &= a + b; \\
\{ a = b, c = b, d = a + b \} \\
e &= a + b; \\
\{ a = b, c = b, d = a + b, e = a + b \} \\
d &= b; \\
\{ a = b, c = b, d = b, e = a + b \} \\
f &= a + b; \\
\{ a = b, c = b, d = b, e = a + b, f = a + b \}
\end{align*}
\]
Common Subexpression Elimination

```
{ }
a = b;
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = d;
{ a = b, c = b, d = a + b, e = a + b }
d = b;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```
Common Subexpression Elimination

```
{ }
  a = b;
  { a = b }
  c = a;
  { a = b, c = b }
  d = a + b;
  { a = b, c = b, d = a + b }
  e = d;
  { a = b, c = b, d = a + b, e = a + b }
  d = b;
  { a = b, c = b, d = b, e = a + b }
  f = a + b;
  { a = b, c = b, d = b, e = a + b, f = a + b }
```

Alex Aiken, Stanford
Common Subexpression Elimination

```
{ }
{ a = b }
c = a;
{ a = b, c = b }
d = a + b;
{ a = b, c = b, d = a + b }
e = d;
{ a = b, c = b, d = a + b, e = a + b }
d = a;
{ a = b, c = b, d = b, e = a + b }
f = a + b;
{ a = b, c = b, d = b, e = a + b, f = a + b }
```

Alex Aiken, Stanford
Common Subexpression Elimination

```plaintext
{  }
a = b;
{  a = b  }
c = a;
{  a = b, c = b  }
d = a + b;
{  a = b, c = b, d = a + b  }
e = d;
{  a = b, c = b, d = a + b, e = a + b  }
d = a;
{  a = b, c = b, d = b, e = a + b  }
f = a + b;
{  a = b, c = b, d = b, e = a + b, f = a + b  }
```

Alex Aiken, Stanford
Common Subexpression Elimination

```
{   }
  a = b;
{   a = b   }
  c = a;
{   a = b, c = b   }
  d = a + b;
{   a = b, c = b, d = a + b   }
  e = d;
{   a = b, c = b, d = a + b, e = a + b   }
  d = a;
{   a = b, c = b, d = b, e = a + b   }
  f = e;
{   a = b, c = b, d = b, e = a + b, f = a + b   }
```

Alex Aiken, Stanford
Common Subexpression Elimination

\[
\begin{align*}
a & = b; \\
c & = a; \\
d & = a + b; \\
e & = d; \\
d & = a; \\
f & = e;
\end{align*}
\]
Live Variables

- The analysis corresponding to dead code elimination is called **liveness analysis**.

- A variable is **live** at a point in a program if later in the program its value will be read before it is written to again.

- Dead code elimination works by computing liveness for each variable, then eliminating assignments to dead variables.

Alex Aiken, Stanford
Computing Live Variables

- To know if a variable will be used at some point, we iterate across the statements in a basic block in reverse order.
- Initially, some small set of values are known to be live (which ones depends on the particular program).
- When we see the statement \( a = b + c \):
  - Just before the statement, \( a \) is not alive, since its value is about to be overwritten.
  - Just before the statement, both \( b \) and \( c \) are alive, since we're about to read their values.
  - *(what if we have \( a = a + b \)?)

Alex Aiken, Stanford
Liveness Analysis

a = b;

c = a;

d = a + b;

e = d;

d = a;

f = e;
Liveness Analysis

\[
\begin{align*}
a &= b; \\
c &= a; \\
d &= a + b; \\
e &= d; \\
d &= a; \\
f &= e; \\
\{ b, d \}
\end{align*}
\]
Liveness Analysis

\[ a = b; \]
\[ c = a; \]
\[ d = a + b; \]
\[ e = d; \]
\[ d = a; \]
\[ \{ b, d, e \} \]
\[ f = e; \]
\[ \{ b, d \} \]

Alex Aiken, Stanford
Liveness Analysis

\[
\begin{align*}
a &= b; \\
c &= a; \\
d &= a + b; \\
e &= d; \\
\{ a, b, e \} \\
d &= a; \\
\{ b, d, e \} \\
f &= e; \\
\{ b, d \}
\end{align*}
\]
Liveness Analysis

\[ a = b; \]
\[ c = a; \]
\[ d = a + b; \]
\[ \{ a, b, d \} \]
\[ e = d; \]
\[ \{ a, b, e \} \]
\[ d = a; \]
\[ \{ b, d, e \} \]
\[ f = e; \]
\[ \{ b, d \} \]
Liveness Analysis

\[ a = b; \]
\[ c = a; \]
\[ \{ a, b \} \]
\[ d = a + b; \]
\[ \{ a, b, d \} \]
\[ e = d; \]
\[ \{ a, b, e \} \]
\[ d = a; \]
\[ \{ b, d, e \} \]
\[ f = e; \]
\[ \{ b, d \} \]
Liveness Analysis

```
a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```
Liveness Analysis

{ b }
a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
Dead Code Elimination

```
{ b }
a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```

Alex Aiken, Stanford
Dead Code Elimination

```
{ b }
a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```

Alex Aiken, Stanford
Dead Code Elimination

{ b }
  a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }

{ b, d }

Alex Aiken, Stanford
Dead Code Elimination

{ b }
a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }

{ b, d }
Dead Code Elimination

{ b }

a = b;

{ a, b }

{ a, b }
d = a + b;

{ a, b, d }
e = d;

{ a, b, e }
d = a;

{ b, d, e }

{ b, d }
Dead Code Elimination

\[
a = b;
\]
\[
d = a + b;
\]
\[
e = d;
\]
\[
d = a;
\]
Liveness Analysis II

\[ a = b; \]

\[ d = a + b; \]

\[ e = d; \]

\[ d = a; \]
Liveness Analysis II

\[ a = b; \]
\[ d = a + b; \]
\[ e = d; \]
\[ d = a; \]
\[ \{ b, d \} \]
Liveness Analysis II

\[ a = b; \]

\[ d = a + b; \]

\[ e = d; \]

\{ a, b \}

\[ d = a; \]

\{ b, d \}
Liveness Analysis II

\[
a = b;
\]

\[
d = a + b;
\]

\[
\{ a, b, d \}
\]

\[
e = d;
\]

\[
\{ a, b \}
\]

\[
d = a;
\]

\[
\{ b, d \}
\]
\[ a = b; \]

\{ a, b \}

d = a + b;
\{ a, b, d \}
e = d;
\{ a, b \}
d = a;
\{ b, d \}
Liveness Analysis II

{ b }
a = b;

{ a, b }
d = a + b;

{ a, b, d }
e = d;

{ a, b }
d = a;

{ b, d }
Dead Code Elimination

{ b }
a = b;

{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b }
d = a;
{ b, d }

Alex Aiken, Stanford
Dead Code Elimination

{ b }
    a = b;

{ a, b }
    d = a + b;
    { a, b, d }
    e = d;

{ a, b }
    d = a;
{ b, d }

Alex Aiken, Stanford
Dead Code Elimination

{ b }
a = b;

{ a, b }
d = a + b;
{ a, b, d }

{ a, b }
d = a;
{ b, d }
Dead Code Elimination

\[ a = b; \]

\[ d = a + b; \]

\[ d = a; \]
Liveness Analysis III

\[ a = b; \]

\[ d = a + b; \]

\[ d = a; \]
Liveness Analysis III

\[ a = b; \]

\[ d = a + b; \]

\[ d = a; \]

\[ \{b, d\} \]
Liveness Analysis III

\[ a = b; \]

\[ d = a + b; \]

\{a, b\}

\[ d = a; \]

\{b, d\}
Liveness Analysis III

\[
\begin{align*}
  a &= b; \\
  \{a, b\} \\
  d &= a + b; \\
  \{a, b\} \\
  d &= a; \\
  \{b, d\}
\end{align*}
\]
Liveness Analysis III

{b}
a = b;

{a, b}
d = a + b;

{a, b}
d = a;

{b, d}
Dead Code Elimination

{b}
a = b;

{a, b}
d = a + b;

{a, b}
d = a;

{b, d}
Dead Code Elimination

{\textcolor{red}{b}}
a = b;

{\textcolor{red}{a, b}}
d = a + b;

{\textcolor{red}{a, b}}
d = a;

{\textcolor{red}{b, d}}
Dead Code Elimination

\{b\}
\[
a = b;
\]

\{a, b\}

\{a, b\}

\[
d = a;
\]
\{b, d\}
Dead Code Elimination

a = b;

d = a;
A Combined Algorithm

Alex Aiken, Stanford
A Combined Algorithm

a = b;
c = a;
d = a + b;
é = d;
d = a;
f = e;

Alex Aiken, Stanford
A Combined Algorithm

\[
\begin{align*}
a &= b; \\
c &= a; \\
d &= a + b; \\
e &= d; \\
d &= a; \\
f &= e; \\
\{b, d\}
\end{align*}
\]
A Combined Algorithm

\[
\begin{align*}
  a &= b; \\
  c &= a; \\
  d &= a + b; \\
  e &= d; \\
  d &= a; \\
  f &= e; \\
  \{b, d\}
\end{align*}
\]
A Combined Algorithm

\[
a = b;\\
c = a;\\
d = a + b;\\
e = d;\\
d = a;\\
\{b, d\}
\]
A Combined Algorithm

\[
\begin{align*}
    a &= b; \\
    c &= a; \\
    d &= a + b; \\
    e &= d; \\
    \{a, b\} \\
    d &= a; \\
    \{b, d\}
\end{align*}
\]
A Combined Algorithm

\[
\begin{align*}
  a &= b; \\
  c &= a; \\
  d &= a + b; \\
  e &= d; \\
  \{a, \ b\} \\
  d &= a; \\
  \{b, \ d\}
\end{align*}
\]
A Combined Algorithm

\[
a = b;
\]

\[
c = a;
\]

\[
d = a + b;
\]

\[
\{a, b\}
\]

\[
d = a;
\]

\[
\{b, d\}
\]
A Combined Algorithm

\[
\begin{align*}
  a &= b; \\
  c &= a; \\
  d &= a + b; \\
  \{a, b\} \\
  d &= a; \\
  \{b, d\}
\end{align*}
\]
A Combined Algorithm

\[
a = b; \\
c = a; \\
\{a, b\} \\
d = a; \\
\{b, d\}
\]
A Combined Algorithm

\[
\begin{align*}
  a &= b; \\
  c &= a; \\
  \{a, b\} \\
  d &= a; \\
  \{b, d\}
\end{align*}
\]
A Combined Algorithm

\[ a = b; \]

\[ \{a, b\} \]

\[ d = a; \]

\[ \{b, d\} \]
A Combined Algorithm

\{b\}
\[ a = b; \]

\{a, b\}
\[ d = a; \]

\{b, d\}
A Combined Algorithm

\[ a = b; \]

\[ d = a; \]